Porting XFS to Linux

The XFS filesystem code on IRIX consists of about 112 thousand lines of code (including comments). We will talk about what happened when we tried to move this to the Linux kernel. Just to make things more interested we also decided to implement filesystem meta data caching in the page cache instead of the buffer cache.

1. Notes

1.1. Original presentation

The original presentation of this talk occurred in room A of the Ottawa Linux Symposium, Ottawa Congress Centre, Ottawa, Ontario, Canada on the 21st of July, 2000 at 10:00 local time. This presentation was given by Steve Lord.

1.2. Presenter bio

Steve is a senior filesystem developer at SGI. Steve graduated from the University of Manchester with a degree in computer science and has been working on Unix systems since 1983 and on filesystems since 1992. A Linux user since 1991, Steve finally got to work on Linux in his day job this year.

1.3. Presentation recording details

This transcript was created using the OLS-supplied recording of the original live presentation. This recording is available from ftp://ftp.linuxsymposium.org/ols2000/2000-07-21_11-18-46_A_64.mp3

The recording has a 64 kb/s bitrate, 32KHz sample rate, mono audio (due to the style of single microphone recording used) and has a file size of 28646784 bytes. The MD5 sum of this file is: eef42fa82717f68f9a01eec6fcf4a579

1.4. Creation of this transcript

1.4.1. Request for corrections

This transcript was not created by a professional transcriptionist; it was created by someone with technical skills and an interest in the presented content. There may be errors found within this transcript; we ask that you report them to using the bug tracking interface described at http://olstrans.sourceforge.net/bugs.php3

1.4.2. Tools used in transcript creation

This transcription was made from the MP3 recording of the original presentation, using XMMS for playback and lyx (with docbook template) for the transcription.

1.4.3. Format of transcript files

The transcribed data should be available in a number of formats so as to provide more ready access to this data to a larger audience. The transcripts will be available in at least HTML, SGML and plain ASCII text formats; other formats may be provided.

1.4.4. Names of people involved with this transcription

This transcript was created by Jacob Moorman of the Marble Horse Free Software Group (whose pages live at http://www.marblehorse.org). He may be reached at roguemtl@marblehorse.org

The primary quality assurance for this document was performed by Stephanie Donovan. She may be reached at sdonovan@achilles.net

1.4.5. Notes related to the use of this document

1.4.6. Ownership of the content within this transcript

1.5. Markup used in this transcript

1.5.1. Time markers

At the end of each paragraph within the body of this transcript, a time offset is listed, corresponding to that point in the MP3 recording of the presentation. This time marker is emphasized (in document formats in which emphasis is supported) and is placed within brackets at the very end of each paragraph. For example, *05m, 30s* states that this paragraph ends at the five-minute, thirty-second mark in the MP3 recording.

1.5.2. Questions and comments from the audience

These recordings were created using a bud microphone attached to the speaker during their presentation. Due to the inherent range limitations of this type of microphone, some of the comments and questions from the audience are unintelligible. In cases where the speaker repeats the audience question, the question shall be omitted and a marker will be left in its place. Events which happen in the audience shall be bracketed, such as: The audience applauds.

Further, in cases where the audience comments or questions are not repeated by the speaker, they shall be included within this transcript and shall be enclosed within double quotes to delineate that the statements come from the audience, not from the speaker.

1.5.3. Editorial notes

The editor of this transcript, the transcriptionist (if you will), and the quality assurance resource who have examined this transcript may each include editorial notes within this transcript. These shall be placed within brackets and shall begin with 'ED:'. For example: ED: The author is referring to sliced cheese, not grated cheese.

1.5.4. Paragraph breaks

The paragraph breaks within this transcript are very much arbitrary; in many cases they represent pauses or breaks in the speech of the speaker. In other cases, they have been inserted to allow for enhanced clarity in the reading of this transcript.

1.5.5. Speech corrections by the speaker

During the course of the talk, the speaker may correct himself or herself. In these cases, the corrected speech will be placed in parenthesis. The reader of this transcript may usually ignore the parenthised sections as they represent corrected speech. For example: My aunt once had (a dog named Spot, sorry) a cat named Cleopatra.

1.5.6. Unintelligible speech

In sections where the speech of the author or audience has been deemed useful, but unintelligible by the transcriptionist or by the quality assurance resource, a marker will be inserted in their places, unintelligible. Several attempts will be made to correct words and phrases of this nature. In cases where the unintelligible words or phrases are clearly not of importance to the meaning and understanding of the sentence, they may be omitted without marker insertion.

2. Transcription of MP3-Recorded Audio

My name is Steve Lord; I'm going to be talking about XFS. Maybe 60-40 between what XFS is in terms of what features it has and what it is and how it works, and what we've had to do to get into Linux.

So I'll talk first a bit about why we're doing this, then I'll move into features of XFS, some of how it works, I'll try not to replicate what went before yesterday, and differences in the interfaces between the filesystem and the rest of the OS between Linux and IRIX, which is where most of the work ends up being. A little bit about pagebufs, which is the work we've been doing for file buffering and metadata buffering, where we are, and where we're going from here. *01m, 00s*

So, first of all, obviously we wanted to be GPL because we want this to go into Linus' tree at some point, we're not exactly sure when that's going to happen, but someday it will happen. Single source base between IRIX and Linux was an initial goal as well. There is a hundred and twenty thousand (120,000) lines, thereabouts, of code in the kernel with XFS. A lot of that's comments. I've not gone in and attempted to say there are this many actual lines of real C code. *01m, 41s*

So we really don't want to have multiple, different versions of this stuff. We want to keep as much of it the same as possible. Then the other important thing is we want to be able to move filesystems backwards and forwards between MIPS IRIX boxes and Linux. *02m, 05s*

First of all, the legal end of things. When SGI announced the open sourcing of XFS, I don't even remember, like, way long time ago, sometime earlyish last year; I really wish we hadn't told the lawyers, because it was horrible. SGI's legal people insisted that we don't do anything which might get the company sued. So basically a group of very brave people in Australia took the source code and they went and collected other source bases, I mean ATT System V, all sorts of stuff, and generated all sorts of tools for doing comparison between the different trees and looking for matches. When they did find one, we had to take it out and replace it. And they found four or five. Very very little. The whole core of XFS didn't look like anybody else's code. Basically, it let them learn the code. It also got rid of two lawyers and it took way too long to do. *03m, 27s*

Source code compatibility. Obviously common code base. The two reasons really; A) the cost for maintaining two; and B) if we have to change everything to make it work on Linux, then chances are we are going to break a lot of it in the process. So it was really fairly key to us that we try to keep the core of the code the same and deal with stuff around the edges. Now unfortunately, at the same time (IRIX didn't have a good, over the last few years) IRIX really didn't have a very good reputation for reliability and they really clamped down on code going into that thing now. Which means, the people who look after that say, no way, no how are you changing our code just so it will work on Linux. So at the moment, we have a lot of changes in our code which aren't in IRIX, maybe they'll go in there someday, but not for now. But we've tried to do things in a way where we can go back. *04m, 34s*

And then, finally, from the external messages we get are like: keep it clean and don't do stuff to Linux which is going to help you but hurts everyone else, basically. And just quickly through this, not very interesting to most of the people in here. The on-disk format is now exactly the same. Once the people in Australia had finished reviewing the code, they then went and did an endian conversion on the thing, because MIPS is big-endian and our first platform, the IA-32, is little-endian. They again did all the work on that end of things, which was very nice because I didn't want to do it. Obviously we need to understand the volume managers from IRIX because if we can just support single disks, it's not very interesting, so that is in progress as a totally separate project. And finally, we'll have to make the system calls look the same, somehow, probably through a library layer. *05m, 34s*

On to some more interesting stuff: what's actually in (earthquake, no, in) XFS, well, first of all, some not-very-useful-these-days file size limitations; 2-to-the-64th power is kind of well 2-to-the-63rd power minus 1 for files, 2-to-the-64th power for filesystems is kind of well, not used practically at the moment. Who knows, someday it may be, but there are restrictions in Linux which pull us down from that. I've managed to create a file with a size of 2-to-the-43rd power minus 1. So the LFS stuff that's out there... this was on Red Hat 6.2, so the userspace LFS stuff does work. Because you can create a file that big, I haven't, I wouldn't recommend counting it, though... (because I just) I mean I don't have that much disk space, I just put a few bytes out at the end. *06m, 40s*

When you make an XFS filesystem, you get to pick the block size of the filesystem. This is the unit of allocation, so when you are allocating directory blocks, when you are extending a file, this is the smallest chunk of stuff it will allocate. That goes anywhere from 512 bytes to 64 kilobytes. Sixty-four kilobytes is just for people who always deal with huge files and there is no point in them messing around with small stuff. The default is four kilobytes. *07m, 13s*

And we are completely extent based. There's a couple of bitmaps laying around one or two places, but free space is managed as extents, in-use space in files is all managed in extents. So extents is basically a tuple of file offset, disk block location and length. Free space (I'll get into a bit of detail there, free space) is actually organized as two B-Trees, one of which is organized by location on disk and one of which is organized by size. So depending on what your first priority is when you allocate space, you can search one way or the other. So if you want to put stuff close to something else, you go to the location search, if you want to get something contiguous, then you go into the size-based one. *08m, 14s*

I'm not going to attempt, I couldn't attempt to draw a picture of what the whole of XFS looks on this, but basically we can have three sub-volumes. You can split your filesystem up into three different sub-components. Data, which is the main part of the filesystem, and that's metadata, file data, free space, everything is in there. The log is just a circular journal. And then we have a real-time subvolume, which is only used for file data, there's no metadata in there at all. IRIX volume managers let you build a volume that has named subvolumes within it and on IRIX you can say 'mkfs' on the volume name and it'll glue everything together. *09m, 13s*

Martin down here, did some work on Linux to let us pass in separate names for different sub-components, but in the case where you are just using a single disk partition, then you don't bother with real-time and basically the log gets put in the middle of the data subvolume. So for the simple case, none of this comes into play. *09m, 41s*

Then the data sub-volume is split into allocation groups. This is the unit of management of space, if you like. Each of these maintains its own free space, its own inode structures, what have you, and you can have one CPU allocating space in one of these while another CPU is freeing space in another one, so it's also a multi-threading thing. (Those are...) If you made a filesystem on a 9 GB drive, they'd probably come in at about 1 GB each. They are limited to a 4 GB size. So when we get to really big filesystems, we get to have thousands of them. That actually gets to be a problem when you want a multi-terabyte filesystem and people are working on that. *10m, 32s*

The other thing it lets us do is have more of them, so if you have a volume manager which lets you add space to a volume, then on-the-fly you can grow the filesystem, create new allocation groups and then people can start putting data in them. When we're actually allocating space, normal case is delayed allocation or, as it seems to be getting called in the Linux community, allocate on flush. *11m, 05s*

When you do a write on XFS, usually there is no transaction. There is in-memory structures which represent the amount of free space available and you just subtract from the amount of free space and tack it on to the file and there is an in-memory representation of an extent which doesn't have a physical disk location yet. Then later on, due to memory pressure or sync activity, or something else, this gets flushed out to disk and at that point we do the allocation, so you coalesce multiple write calls into a single allocate request. This helps a lot for creating contiguous files. *11m, 50s*

We also support preallocation, so if you know in advance what your application is going to be using, in terms of space, then you can pre-layout the space in advance and get all of the transactions out of the way, and you can just push the data straight into it when you actually write it. It tends to go together with direct I/O, where you're moving directly between the user memory and the device and not going through the buffer cache. *12m, 22s*

The other thing here is the allocator is fairly smart; it knows about striping. So if you have a striped filesystem, it attempts to lay out extents on stripe boundaries; that tends to give you better throughput. If you have four drives on four controllers striped correctly, then you could basically have all four of them going full speed ahead at once and file extents and inodes get laid out on those boundaries in a striped filesystem. There's also other stuff in there for placement. For the simple filesystem case, it tries to keep file data close to the inode and file inodes in the directory close to the directory, but it spreads directories around, so it tries to put a subdirectory somewhere else, just so that you don't end up with everything in one part of the filesystem. *13m, 29s*

We, too, dynamically allocate inodes. Usually it is two filesystem blocks worth and because they are variable size that gives you a different number of inodes depending on how you made the filesystem. They are anything from 256 bytes to 4 KB each. There is a fixed core which looks fairly normal, then the other parts of it gets used for various things. You can put file extents in there, to avoid having indirect blocks for having that. You can put directory entries in there for small directories. Symlinks go in there. You can put extended attributes in there. These are basically name equals value text type objects. The code is pretty much the directory code. *14m, 29s*

The speaker calls on an audience member for question.

Is the sizing guide number per filesystem or per cluster?

It's per filesystem. It's a mkfs option, so you can't change it on the fly. It's really only used by people who know in advance what type of data they are going to put in there and they want to, for instance people who are extremely concerned about I/O speed don't want to have heads flying around the place doing block lookups for bits of metadata, so they try and organize things so a lot of the directory contents and extents would stay in the inode, rather than being scattered around. That way, more of the head seeking time is spent going through the data, rather than your metadata. *15m, 18s*

The speaker calls on an audience member for question.

This extended attribute call... Ted Ts'o was saying yesterday about how bad an idea it was to have data forks and resource forks. What difficulties have you encountered over the years in IRIX in having extended attribute calls?

To be honest, they don't get used by applications in IRIX too much. If people had been writing GNOME or something on there, then maybe they would've started sneaking stuff in there. Most of the use that is actually made of them is these things down the bottom which are system managed objects, so access control lists, capabilities; and data migration information, where there is a tertiary storage system sitting behind the filesystem where the extents are no longer there, the data is actually sitting on a tape somewhere and when you actually come to want to look at it, the system has to go get it for you. That code is actually starting to go into our open source tree at the moment, so DMAPI support is coming. *16m, 41s*

This is just a picture of that, so as I said, fixed inode core and then there are these two variable-sized chunks of data in there, and the attribute hook is usually empty. Most of the time we don't actually have anything down there. So all the space gets taken up by directory contents, etc. and then once stuff doesn't fit in there, it starts migrating out into indirect block structures. *17m, 10s*

Okay, I already said B-Trees, so directories are organized that way, extents are organized that way, free space, those extended attributes, if you have too many of them, so on and so forth. We are extremely multithreaded. IRIX is currently running on up to 512 CPU boxes, so that's single system image across 512 CPUs. XFS runs on that, it also runs on this laptop here. And as I said before, it can handle multiple parallel applications. *17m, 51s*

Now this real-time subsystem is something which the code is there but we're not doing anything with it at the moment. That lets you do a much more deterministic type of I/O. The whole point of real-time is, you need to be able to give guarantees about how long it takes to do something. So the real-time subsystem actually has (its whole,) it's own allocator, it does its own space allocation, it's a much more deterministic algorithm that's used, and it's also extremely difficult to fragment files out there. We have to actually, usually change the code to make it fragment files, but what goes with that is, it's wasteful of space. I mean, this is for big stuff; this is for MP3s, I suppose, in this audience. *18m, 48s*

Just a quick run-through of what happens to a directory. You create a directory with nothing in it. You start putting entries in it. They live inside the inode. Once it gets too big for that, we move it out into a single block of directory entries and once you go beyond that, it starts building a B-Tree. The actual directory entries only live at the bottom and everything above that is hash value, block address pairs, so you hash your name and can walk down the levels of the tree until you find the specific entry at the bottom. And for the sake of stuff like getdents(), all the leaves at the bottom are connected together, so you can just walk left or right across the thing. And if you start removing stuff from a directory, that actually does collapse back again. So you can actually go all the way back from a B-Tree, back to everything being inside an inode. And extents do exactly the same thing. *20m, 01s*

Okay, so on to journaling, which I haven't said anything about yet. Another feature of XFS is that it is another one of these journaling filesystems. So we do the same thing as everybody else. Basically (we do,) we write to the log, then we write the metadata out, so we have the right ordering requirements that one must happen before the other. Transactions themselves are asynchronous so you could actually complete... most transactions complete without having to write anything, so they get batched up the same way as EXT3, which was talked about yesterday. And let me see, we are different from EXT3 in that we only log deltas of what's changed. So if you manipulate a directory block, and you happen to just put one entry into it, then all that happens is you log the chunk around that; it chops it up into fixed-size pieces, I think. So there's actually quite a lot of compression that goes on in the data. A number of the structures that we log aren't actually copies of blocks. (There are) inodes go out differently; freeing stuff is dealt with differently, so the log isn't just a copy of the metadata. (And then, whoops, oh well. The speaker laughs.) *21m, 40s*

Okay, the other aspect here is that you mount the filesystem; it always looks at the log when you mount the filesystem, but if it finds that the last thing in the log is an unmount record then that's it. If it's not an unmount record, then it has to go find the head and tail of the log and replay it. And again, as with EXT3, and I'm sure JFS, this is a multi-stage process. You have to, there is stuff in there that you don't want to really replay because something else happened later which re-used those disk blocks. But it's usually pretty fast.

There is a question from an audience member; it is repeated a second time, more audibly. The initial question and answer have been omitted, as they are repeated in more detail.

I asked whether this was done in the mount code instead of a userspace application.

Yes, this is within the mount code in the kernel. *22m, 40s*

So, the next several slides are actually a kind of sequence of events of what happens as we go through a transaction and there's a lot of steps missed out here, because it's really a lot more complicated than this. So the first few things you do are reserve log space, and that's where you get stuck if things are busy. I mean basically you're not allowed to do anything until there's space in the log to do the worst-case version of what you're trying to do. (There's a lot...) Doing something like a directory update, the range of amounts of data involved is very large. From the worst case of just updating something in the inode to having a five-level-deep B-Tree and you're splitting the thing and it's going all the way up and you've got to go and allocate some free space and that involves another B-Tree split, so that gets big. So once that's happened, then you obtain lock and modify all your metadata, so that's what (other filesystems, so that's what) EXT2 does. That little bit there. *23m, 50s*

The next thing after that is when we commit the transaction, the deltas get copied into these in-call log buffers. There's several of these; it's a mount option how many of these you get. That's basically it for the transaction. We pin the metadata in memory. This is the XFS terminology for this. Everybody has their own journaling technology, but we basically stick a pin through the stuff and nothing will write it out to disk until we unpin it and then we unlock it which means that other transactions can come along and again modify the metadata (again). *24m, 38s*

So the next step is, this stuff goes out to disk, triggered by a number of different events in the system. One is we run out of in-call log space. Those buffers are 32 kilobytes each and I think two is the default. You can have up to eight. And probably for really large systems, we need to increase that, but one transaction can fill more than all of them. It can actually involve going around, through the buffers multiple times, in those worst case scenarios. Usually several transactions will fit in one buffer. So running out of space, synchronous transactions, for example, and O_SYNC write, which allocates space, you have to push the transaction out to disk at that point and finally, just background sync activity, so like the once every five seconds kind of thing. *25m, 43s*

Once that write completes, we actually need to keep track of all the metadata which is in memory, modified but not on disk yet. So there's this thing called the active item list which is basically a chain of all that metadata. And we can now unpin it, so now the metadata could be written to disk. Obviously this list is (organized by... it's) time organized, which means it's log position organized. *26m, 18s*

Now we push the metadata out, so once again flush, running out of memory, or we don't have any disk log space left. Basically, you're dirtying metadata faster than it's getting flushed out by sync. In that case we need to do what we call tail pushing. Which is, you go find the oldest metadata and you force a flush on it; you force it out to disk. And at that point, it goes out to disk; when that I/O completes, it gets taken off this active item list. And if it happened to be at the end, we can move the tail of the log to free up more log space which means the next guy can come through the transaction, the start of the transaction can be carried on. *27m, 14s*

There's other things going on in here. If we happen to log a piece of metadata and then we log it again before it's gone out to disk, then we just re-log it back into the log again. We move it (to the tail of the active item list, sorry) to the head of the active item list, if it happened to be the tail, we free some log space. So again, as Stephen Tweedie mentioned yesterday, quite a bit of commonly-used metadata just keeps floating around the log and doesn't go to disk very often at all. *27m, 51s*

Okay, on to what we had to do going to Linux. So IRIX is vnode-based, Linux isn't. And IRIX has a buffer cache interface where you get to mess with variable-sized buffers rather than the fixed-sized buffer heads which Linux uses. And lock device interface looks different. System call interface looks different. And then there's some 64-bit'edness, if you want to call it that. So, because we wanted to keep as much of the code as same as possible, we introduced this glue layer, which is basically mapping from the Linux VFS operations to the vnode operations, if you like. Most of what it's doing is argument translation, but it lets us keep a lot more of the structures looking the same. *28m, 58s*

The other thing it gives us is, it can support stacking. You could stack different sets of operations, one on top of the other. And this is different stacking than other software available for Linux gives you, because this is on a per-inode basis. So we can say we want to stack that one, not everything in the whole filesystem. We put this in there for CXFS, which is a clustered version of XFS, which needs to interpose itself between the user and the filesystem. *29m, 37s*

The other thing I have to say here is that the Linux VFS does more than the IRIX VFS. There's a lot more common. Over time, I've watched a lot more code move out of the filesystems and up to the VFS layer. And we still have duplication; things like renaming a directory into a child of itself is one example of a thing you don't want to happen in the filesystem and in IRIX, that's handled by the filesystems. In Linux, it's handled by the VFS. And now in XFS on Linux, it's handled by the VFS layer, so we are working our way through those as we find them. I'm sure there's several left. *30m, 25s*

There is a question from an audience member.

The stacking you talk about... do you intend to try and put it into the VFS?

Haven't really though about that one, to be honest. Another option... at the moment I can't remember the name of the package... The response from the audience member is inaudible. I mean possibly it's an extension to FIST, that would be another way to go with this because FIST, as I understand it, is – you get the option of stacking everywhere or stacking nowhere, basically. *31m, 03s*

So this is a picture of what that looks like. Over to the left, we have the Linux world; the file operations, the inode operations... I didn't show the dir entry operations. And what's left of our vnode is actually the filesystem dependent portion of the Linux inode in there. And then that points at the behavior which is in the same structure as the XFS inode itself and its the behavior which supports this stacking concept, so you can layer those; they chain together. *31m, 44s*

Missing system calls. There's all these twiddly bits in XFS which you can't really get to through the standard Linux system call interface. Here's a few examples of that. Space pre-allocation; you need to be able to put space into a file and take it out again. There is a POSIX spec for this one there, so I suspect that's the way that one will go. Extended attributes; there's no system calls for those yet. We're actually putting the IRIX ones into our tree, but it sounds like there's going to be some sort of interface in Linux in the near future and obviously that's the way that will end up being done. But for now, we need a way of exercising the code in the kernel , so we have to put something in, in the short term. *32m, 42s*

There's other extensions to that. If we want to do the access control lists and mandatory access control and capabilities and stuff, there are system call interfaces for that, but there is also a boat-load of userspace stuff that goes with that type of thing. I think all of the kernel, the actual MAC-access checking, code is not in our tree. But we have the hooks in the filesystem and (that code is actually being published by) the code that XFS uses in IRIX is being published by SGI elsewhere. So the kernel code is just a matter of gluing things together. *33m, 21s*

And the other thing here is real-time inode attributes. If you want to use the real-time subsystem then you have to tell the filesystem about it and you need to say things like, What is the unit of allocation? You might want to say, Whenever I allocate this space in this file, allocate in chunks of half-a-megabyte. That type of operation. *33m, 46s*

This is where things get a little bit more hairy. On the IA-32, the inode number which is used by most of the kernel to identify a particular file is a 32-bit quantity. Well in XFS, our inode numbers are 64 bits. The inode number is actually an encoding of the disk location. It doesn't become a problem until the filesystem is bigger than two terabytes, so at the moment we haven't dealt with this. *34m, 25s*

But Al Viro was suggesting we use some sort of dynamic generation of numbers in place of our inode, so we'd have fake inodes on top of the real inode numbers. To be honest, I don't like that. These things really have to be permanent. People can stat() them out of the kernel and record them somewhere else and expect them to be the same later. So they need to really map onto exactly what the kernel is using. So if anyone has got any bright ideas... *35m, 02s*

(The other one is...) The next one is device addressability. Two terabytes (2 TB) is still the block device addressability limit. Again, for large systems we'd like to be able to go beyond that and hopefully the kiobuf interface is going to give us ways of dealing with that. *35m, 25s*

Another one which actually came up yesterday, talking to the JFS guys, is directory offsets. These, again, can be a 64-bit quantity. They're used on the getdents() system call, so when getdents() gives you back data from the kernel, it gives you an offset to go with it. And you can seek, in theory, to that offset and go back to where you were. *35m, 59s*

Now glibc has a different-sized dirent structure than the kernel does and it does these interesting things where it takes the user's buffer and says, Aha! I know that the kernel structure size is different, so I am actually going to use a heuristic to say: the user asked for this much and I'm going to ask the kernel for a different amount because I know what the kernel will give me won't fit in the user's buffer, if I just blindly ask – or potentially won't fit. Now I can't remember, there's one field which is missing from one and not the other; I can't remember what it is. If this heuristic doesn't work, it does a seek back and tries again. *36m, 44s*

Now that seek only remembers 32 bits of information which doesn't really work in all circumstances and there's other places where the 32 bits becomes an issue. XFS actually has two versions of its directory structure to deal with issues like this. In the second, what we call V2 directories, the problem doesn't actually show up until directories get really, really, really big, like two-to-the-thirty-two (2-to-the-32nd power). The other one, the directory offset is actually a hash value, it doesn't just increment – it wanders all over the place, but all sixty-four bits are relevant. That V1 directory format is going to be an issue on Linux unless we can get a patch into glibc. And that doesn't seem to be very easy to do. The speaker laughs. *37m, 37s*

Okay, so buffering. Most of the filesystems in Linux, when they actually want to do I/O, they speak buffer heads and buffer heads you get to pick one size. And if you don't do I/O in that size, then the block interface throws it back at you and says; don't do that. Now in XFS, I said we have a variable block size at mkfs, but that's not the real problem. The problem is that we have some structures on the disk which are always 512 bytes and we need to be able to do I/O on them, on their own. For the write ordering, we can't just write the whole 4K block which happens to contain the superblock – we must just write the superblock and not the thing next to it which might be pinned in memory. So one option would be we do everything in 512 byte pieces. Well, you get an awful lot of buffer heads when you do things that way and it's pretty expensive. *38m, 40s*

Now obviously we need to be able to control the write ordering, which doesn't work with the regular buffer interface and everyone's coming up with their own way of dealing with that at the moment. And then there was this long-term goal we sort of seem to have heard somewhere about moving more and more stuff into the page cache, and maybe making buffer heads go away altogether. So we decided to make life interesting for ourselves and do this. So XFS is actually caching everything in pages and the only times we use buffer heads at the moment are if we are talking to a device which we need to use them on, so they're created temporarily – we do the I/O, we tear them down. And we are currently using them for some of the write-behind-the-file data. *39m, 38s*

So we created this thing called pagebufs. Pagebufs is really two groups of stuff. So we cache everything in the filesystem in pages, so most filesystems in 2.4, there's an address space in the inode which caches file data, so file data is just held there. We have a special inode which is for the metadata of the filesystem, with an address space in it. We probably should just have an address space and not the inode. That's where the metadata lives. Now we use kiobufs underneath this, so a pagebufs isn't understood by the I/O layer, but the kiobuf is, or will be. So currently it's working for SCSI. There's been some discussion, but Jens is going to do it for IDE. And Martin, he's over here, he's working on LVM, and then maybe he'll try MD. The speaker laughs. The other thing in here is we do locking on pagebufs, so our unit of locking isn't the buffer head or the page, it's the pagebuf. I mean page locking still comes into things, but we can lock things of different sizes. *41m, 16s*

So, I said there are two parallel parts to this. One is file I/O, as I said, we're extent based. The set of operations you get for doing I/O to a file are block-based. So when you actually want to ask for space with the interface that's available, you go to the filesystem and say 'give me the address for this block' and then 'give me the address for the next block', and then the next block, and so on. It doesn't really fit with XFS' model of doing things, so we actually have a parallel set of code, including read and write, for using an extent-based allocator interface. *42m, 02s*

And it understands delayed allocate. We actually currently have our own page cleaner in there, wandering around looking at pages that have this delayed allocate flag in there and calling back into pagebuf and saying: okay, time to write this. At which point, we can use the extent-based allocator interface to do I/O clustering, so we can actually go and say: what's next to this? It's very easy to go to the allocator and say, given this particular offset in the file, give me the range of things which is physically contiguous with it on disk. And from that, we can actually generate a much bigger I/O request. So basically we use the smarts which the filesystem has, rather than relying on the elevator code. I'm not trying to say the elevator code is no good. The speaker laughs. It's just, we're doing everything twice that way. This page cleaner should go away and be replaced with regular parts of the VM system – there are discussions going on about that. (And I think I said everything in here. Okay.) At the moment again our metadata, we're doing our own management of when it gets written and that's something else which should move out of this out into the regular – the rest of the system. *43m, 40s*

So, where are we? Despite what the website says, this filesystem does actually work. For a long time, it said this is pre-alpha code, don't touch it with a large pole, basically. I think now it says it is pre-beta code. The speaker laughs. But I use it every day. Okay, unlike Stephen Tweedie, I haven't installed it all over my laptop. I'll do that sometime. I will admit, we do have a bug at the moment which causes us not to flush inodes when we should do; I have to put some fairly heavy pressure on the system to make that happen and once I stop going to conferences I'll go fix that. *44m, 31s*

So that's IA-32. We actually have this up and running on MIPS as well, so it's running on a big- and little-endian system. We can mount IRIX disks with that. The other ports are probably going to take a little while yet. The userspace code in XFS, for historical reasons, is the kernel code. It's built two different ways. The speaker looks around. Nope, no Dave Miller in here. Dave Miller – the first day we announced all the code was there, he downloaded it, attempted to build it on the SPARC, and threw up in horror; sent us a couple patches and said no way, no how is this ever going to work. So what actually has been happening is the userspace code is still the kernel code, but it is starting to be broken apart. And fairly soon now, we should have a userspace which is a separate copy of the code and we can start cleaning it up. So that should start to make some of the other ports less painful. *45m, 42s*

Most of the utilities are there now, so mkfs... repair, which is for when things actually do go wrong, because with journal filesystems, it doesn't always work. No matter what people say, it doesn't always work; there's I/O errors, all sorts of stuff can happen; developers getting it wrong, you know they tend to screw up filesystems. That's there... xfs_db is basically an editor, you can go in and if you can understand the command interface on the thing, you can actually go in and walk around the filesystem and mess with bits of it. Dump/restore, this slide has it, up until this morning was a work in progress; last night they actually did succeed in doing dump and restore. So I don't think that code is out on our website yet. I think they've been holding that one back because it wasn't at all functional. But I expect that to show up in the next day or so. *46m, 45s*

There is a question from an audience member.

Is it your intention to keep the on-disk compatible between the Linux port of XFS and the original IRIX port of XFS?

Yeah. It's just not worth it to have two different ones. Initially, we were worried with the overhead of byte-swapping and that it was going to cost us too much and initially we were little-endian, because it was easier to get the filesystem going without having to worry about the endian issues. And it was going to be run-time switchable, but first thing you have to do is to make it work the other way around, so do the endian flipping and they measured it and it wasn't worth bothering with and now it's all big-endian. XFS already has several different versions, just like most filesystems have, and there's backwards and forwards compatibility stuff in there. I expect over time other bits and pieces will show up. *47m, 54s*

There's other stuff out there that goes with these tools; somebody just started working on the defragmenter. IRIX has a defragmenter which is an online beast and someone outside of SGI just said hey, I'd like to work on that, so that's cool. *48m, 12s*

There is a question from an audience member.

Is there a resizer?

There is a grower, there isn't a shrinker.

That's fair enough.

Shrinking is much harder. You kind of have to design a filesystem from the start to cope with that. *48m, 28s*

There is a comment from an audience member.

Talking with Stephen, it's an interesting hack, but for those of us contemplating actually running servers on this, the grower is far more important.

I don't think that's there yet, because the last item on this says volume manager interface needs work.

There is a comment from another SGI XFS team member in the audience.

I think that meant the client at some point, but think we might get it done. It's just a matter of getting the API done on that.

Yeah. The kernel userspace interface is very minimal, it's just like, here's some extra space now go use it. But on the volume manager front, there's that, there's finding out where the heck the stripes are. While the filesystem knows about them, you have to get the information from the volume manager into the filesystem and that's not quite there yet. *49m, 20s*

There is an unintelligible question.

Yeah, you can actually, if you go and do your research and figure out how the heck the thing is laid out, you can actually make a filesystem and explicitly specify the options.

There is an unintelligible comment regarding LVM.

Check your code in. The speaker laughs.

Seems to be the general theme around here. The speaker laughs. *49m, 43s*

Okay, so where do we go from here... Okay, there's always performance work to do. It's actually pretty reasonable at the moment. I think for bonnie, we're doing about twice EXT2, for like if you throw multiple times the amount of memory you have in your system into one file, so that you write a gigabyte file with bonnie or something. Is that right, unintelligible?

There is an unintelligible response

But there's things we can do yet. There's definitely things we can do yet. *50m, 34s*

There is a question from an audience member.

What is XFS' performance on many, many small files?

Depends on what you do with the files.

The explanation continues.

Let me explain, my department is the biology department at Duke University. I've got a herd of gene sequencers. The gene sequence is ten zillion files, all averaging about 700 bytes in length.

You probably want this guy over here. No, I'm not afraid to stand up here and say that reiserfs is going to be better for some things, XFS is going to be better for others. I don't regard this as a competition. There's different horses for different courses, basically. XFS is going to do okay, but I suspect other filesystems will do better. *51m, 29s*

There is a question from an audience member. The question is somewhat inaudible, but was related to the amount of changes in XFS, between the open source tree and the original IRIX code.

Things we changed in XFS itself, well the interface to buffers was the largest change, we basically encapsulated the IRIX buffer interface and the pagebuf interface behind the set of macros, so we had to go all over the code changing where XFS dived in and out of buffer structures directly. So there was that, but it was pretty much a one-for-one exchange. Because of the way inodes are managed in IRIX compared to how vnodes; sorry, inodes in Linux versus vnodes in IRIX. We have some changes in the inode tear down end of things, which is where the bug is. Apart from that, the changes are basically taking stuff out which is done by the VFS layer and changing system call interfaces at the top. The most important thing is getting the caching behavior exactly right. The core of the code is the same. The algorithms are the same, we haven't had to do drastic surgery in there. So yeah, that was one of the key reasons for keeping the code the same; it's been tested a lot and we don't want to mess with it. *53m, 25s*

So, okay, what else on here... pagebuf and Linux VM integration is going to happen; we're not exactly sure what's going to come out the end of it, in terms of what pagebufs end up looking like, but we are going to get more tightly integrated that end, which will help with general system behavior. We're not too... I can make XFS drag a system into the floor basically, but it's because it's thrashing the disk backwards and forwards just because I'm putting heavy stress on it. But I'm sure you can do that with other filesystems as well. But generally, (it doesn't kill) it doesn't grab all the memory in the system or anything like that. It's reasonably well behaved, but we can do better. *54m, 14s*

Real-time, I've said the code is there. The thing that isn't there is the support for it. There's this thing called guaranteed rate I/O, which is basically a reservation mechanism where you say: I want two megabytes per second for two minutes. And this userspace code is responsible for managing that and saying either you've got it or you didn't get it. And there's assorted hooks down in the I/O system in IRIX for maintaining those guarantees. Whether those would ever go into general Linux, (I don't) I'm not sure at all; I doubt it. *54m, 52s*

Then there's the volume manager end of things. It's still a work in progress. So that's purely a portability thing. We'll have Linux volume manager support. And then, once I've got all that done, or we've got all that done, it's not just me, we have to go do it again for CXFS, which is the clustered version of XFS. And that's going to be really interesting. *55m, 27s*

There is a question from an audience member.

Is the problem unintelligible or are there issues with Linux? unintelligible

Well, CXFS does caching. I don't know if you were in the GFS talk. There still is one server; there's still one place where XFS is running and all the other machines use fibre-channel to get to the disk, but they do it... they do direct I/O between their memory and the file data blocks. All metadata is mediated through the server, but CXFS lets you do things like move the server while the system is running. Which means you've got to do lots of stuff with stopping people getting through the filesystem interface, holding them up, moving stuff out from under them, putting other stuff (in the place) in its place, and a lot of that is very filesystem interface specific. That's one of the areas where things are quite different. *56m, 31s*

There is a question from an audience member.

Are you going to open source the CXFS part? Because last time I talked to your engineers, there was some discussion about that.

Actually, I doubt it. One day, probably. But currently, it's something we can actually make money on, and SGI as you all know probably needs money. The speaker laughs. So, probably that's not going to be open source, which is going to make it really interesting, it's probably going to get tied to specific kernel versions and all sorts of weird stuff, but it'll be... don't hold your breath. *57m, 10s*

There is a question from an audience member.

Is the volume manager open source?

I believe it will be, yes. And there's two aspects to that. One aspect is that you need to be able to understand the labels on the disk and that's not too hard to do, to implement from scratch. XVM understands clusters, is one of the other reasons for doing it. And that's a lot of management code, but the actual labelling of disks could be done in a much smaller piece of code, I suspect. But yes, it will be open source. *57m, 57s*

There is a comment from an audience member.

unintelligible The main incentive for porting XVM would be CXFS...

And just so people, SGI customers, can move disk farms around between systems.

An audience member requests that the question be restated.

Okay, so the Linux LVM plus MD provides most of the functionality we have in XVM, so there's no real point in porting that to get another volume manager. The intention for porting the volume manager would be CXFS and being able to move this farm from IRIX boxes to Linux boxes. *58m, 41s*

You never know, maybe you guys do some part of it better.

We'll see. *58m, 49s*

It supports more ways of gluing disks together than the open source stuff in Linux does at the moment. You can stack things three ways from Sunday or whatever you want to call it. Which doesn't always make that much sense, but you can do it. So you know you can stripe on top of RAID, on top of stripe on top of ... basically, it's a complete stacking implementation.

Any more questions?

Okay. Thank you.

The audience applauds. *59m, 40s*

3. Additional resources

3.1. Presentation materials

The slides used for this presentation are available at
http://oss.sgi.com/projects/xfs/papers/ols2000

3.2. XFS

Project information regarding the XFS porting effort, as well as source code for the Linux XFS port are available at http://oss.sgi.com/projects/xfs/